

*INF 321*  
*Les fonctions*

Eric Goubault

Cours 2

13 mai 2011

## ON A VU:

- Le noyau impératif
- Une première approche des programmes Java

## ON VA VOIR:

- Une première structuration du code: les fonctions
- Quelles fonctions peut-on *calculer* avec le noyau impératif?

# POURQUOI DES FONCTIONS?

## JAVA, C

```
/* Calcul de x */  
...  
  if (x<0) sgnx=-1;  
  if (x==0) sgnx=0; else sgnx=1;  
...  
/* Calcul de y */  
  if (y<0) sgny=-1;  
  if (y==0) sgny=0; else sgny=1;
```

## ET?

- Pas pratique, redite, cause potentielle d'erreur
- → Structurer par des appels à des fonctions!
- Plus profond que cela...: récursion et langages fonctionnels (Caml): "Functions as first-class citizens" (Christopher Strachey, ~1960)

## UNE FONCTION PEUT:

- prendre en compte des données: *arguments*
- retourner une valeur: *le retour de la fonction*
- effectuer une action (afficher quelque chose, modifier la mémoire): *effet de bord*

Dans la plupart des langages, on indique le type des arguments et du retour de fonction (sauf en général dans Caml, où il y a inférence de types - on y revient plus tard), par exemple `int`, `float` etc.

## *Procédure:* FONCTION SANS RETOUR DE VALEUR

- Ne pas retourner de valeur = retourner valeur dans un type *vide*
- en Java, et C, mot clé `void`
- en Caml, type spécial `unit`

## JAVA, (C)

```
static int signe(int x) {  
    if (x<0) ...
```

(retour de fonction, après...)

## CAML

```
let signe x = if (x<0) then ...
```

Aussi fonctions prédéfinies, par exemple *paquetage* (“module”, “bibliothèque”) de fonctions mathématiques `Math`, en particulier, fonction racine carrée: `Math.sqrt(...)`

## JAVA, C

```
/* Calcul de x */  
...  
sgnx = signe(x);  
/* Calcul de y */  
...  
sgny = signe(y);  
...
```

Remarque: les arguments d'une fonction sont évalués avant de démarrer le *code* de la fonction (on aurait pu faire `signe(expr donnant x)`)

## CAML

```
...  
in  
let sgnx = signe x in ... ;;
```

# LE RETOUR DE FONCTION

## JAVA

```
static int signe ([ final ] int x) {  
    if (x<0) return -1;  
    if (x==0) return 0;  
    return 1; }
```

(on revient sur static plus tard)

## C

```
int signe ([ const ] int x) {  
    if (x<0) return -1;  
    if (x==0) return 0; return 1; }
```

## CAML

```
let signe x = if (x<0) then -1.0 else  
              if (x==0) then 0.0 else 1.0;;
```

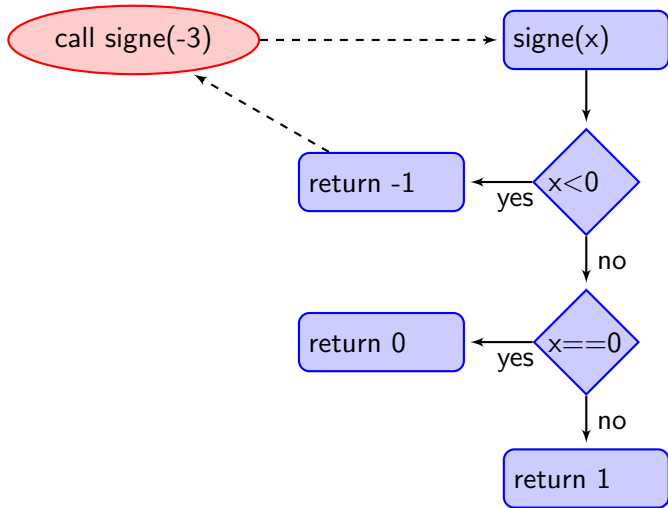
(pas besoin de définir le type: *inférence de types!*)

## JAVA

```
static int signe(int x) {  
    if (x<0) return -1;  
    if (x==0) return 0;  
    return 1;  
}
```

return interrompt le déroulement de la fonction

return interrompt le déroulement de la fonction:



# EN FAIT, VOUS CONNAISSEZ CELA PAR COEUR...

## IDENTITÉ DE BEZOUT

$$a \wedge b = 1 \Rightarrow (\exists p, q, ap + bq = 1)$$

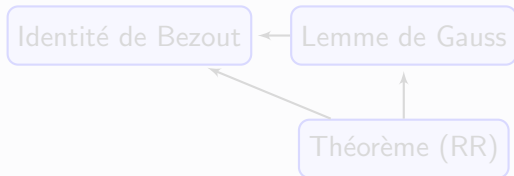
## LEMME DE GAUSS

$$(a \mid bc \ \& \ a \wedge b = 1) \Rightarrow a \mid c$$

## THÉORÈME DE REPRÉSENTATION DES RATIONNELS (RR)

$$\forall r \in \mathbb{Q}, \exists ! p, q, p \wedge q = 1, q > 0, \text{ tel que } r = \frac{p}{q}$$

## PREUVE(S)



# EN FAIT, VOUS CONNAISSEZ CELA PAR COEUR...

## IDENTITÉ DE BEZOUT

$$a \wedge b = 1 \Rightarrow (\exists p, q, ap + bq = 1)$$

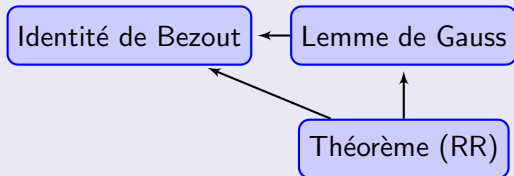
## LEMME DE GAUSS

$$(a \mid bc \ \& \ a \wedge b = 1) \Rightarrow a \mid c$$

## THÉORÈME DE REPRÉSENTATION DES RATIONNELS (RR)

$$\forall r \in \mathbb{Q}, \exists ! p, q, p \wedge q = 1, q > 0, \text{ tel que } r = \frac{p}{q}$$

## PREUVE(S)



## PREUVE(S)

### Identité de Bezout

## PSEUDO-CODE CORRESPONDANT - *preuve constructive*

Par l'algorithme d'Euclide étendu:

```
Bezout(a, b) retourne p,q,r tels que  $r = \text{PGCD}(a,b)$  et  $pa + qb = r$  {  
   $(r1, r2) = (a, b)$ ;  $(u1, u2) = (1, 0)$ ;  $(v1, v2) = (0, 1)$ ;  
  tant que  $(r2 \neq 0)$  {  
     $q = r1 / r2$ ;  
     $(r1, r2) = (r2, r1 \% r2)$ ;  
     $(u1, u2) = (u2, u1 - q * u2)$ ;  
     $(v1, v2) = (v2, v1 - q * v2)$ ;  
  }  
  retourne  $(u1, v1, r1)$ ;  
}
```

## PREUVE(S)

Identité de Bezout



Lemme de Gauss

## PSEUDO-CODE

```
check_divide(x, y) retourne k tel que  $kx=y$ 
    si x divise y
    sinon erreur { ... }

Gauss(a, b, c) retourne u tel que  $ua=c$  {
  k = check_divide(a,b*c); //  $ka=bc$ 
  (p,q,r) = Bezout(a,b); //  $pa+qb=r$ 
  si (r == 1) { //  $pa+qb=1$ 
    retourne  $p*c+q*k$ ; //  $(pc+qk)a=cpa+cqb=c(pa+qb)=c$ 
  }
  sinon erreur();
}
```

Plus profond qu'il n'y paraît...Curry-Howard (un peu en cours 4)

# COMMENT SONT PASSÉS LES ARGUMENTS?

## EXEMPLE

```
class troisVerres {  
    public static void main(String [] args) {  
        int a=1;  
        int b=2;  
        int c=a;  
        a=b;  
        b=c;  
        System.out.println("a="+a);  
        System.out.println("b="+b);  
    }  
}
```

java troisVerres donne a=2, b=1, OK!

```
class troisVerres {  
    static void swap(int x, int y) {  
        int z=x; x=y; y=z; }  
  
    public static void main(String [] args) {  
        int a=1;  
        int b=2;  
        swap(a,b);  
        System.out.println("a="+a);  
        System.out.println("b="+b); }  
}
```

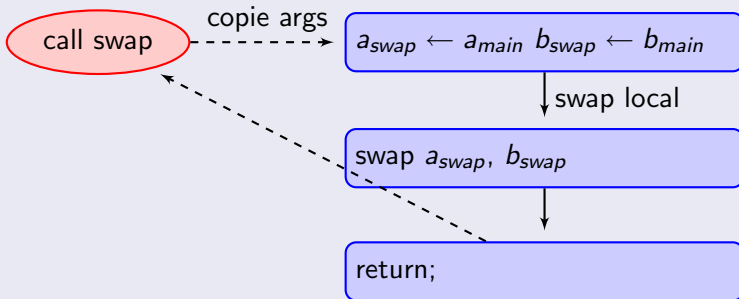


java troisVerres donne a=1, b=2, pas OK!

Pourquoi?

## PASSAGE PAR *valeur*

- En Java, C, Caml, on n'a par défaut que le *passage par valeur* (des variables scalaires)
- La fonction travaille dans ce cas sur une copie des variables passées en paramètre



La modification n'a aucune portée au delà de la fonction!

## VARIABLE GLOBALE

```
class C { static int x; x=3; x=0;
```

x vaut 0 à la fin de l'exécution

## VARIABLE LOCALE

```
static void reset () { int x=0; }  
public static void main (String args []) {  
    int x; x=3; reset (); ...
```

x recouvre deux variables en fait:

- une variable locale à `main`, qui vaut 3 et n'est pas modifiée
- une variable locale à la fonction `reset`, qui vaut 0, et qui n'est connue que dans le corps de la fonction `reset`
- cette variable locale *cache* la variable `x` de `main` dans le corps de la fonction `reset`, mais ne l'écrase pas (location distincte)

```
... main ...
```

```
int x; x=3;
```

$x_{main} : 3$

# ILLUSTRATION (2)

```
... reset ...
```

```
int x; x=0;
```

$x_{main} : 3$

$x_{reset} : 0$

# ILLUSTRATION (3)

fin du main

$x_{main} : 3$

$x_{reset} : 0$

## EXEMPLE

```
static void tab_plus_un(int [] y){
    int n = y.length;
    for (int i = 0; i < n; i=i+1)
        y[i] = y[i]+1; }
public static void main(String [] args) {
    int [] x = new int [5];
    for (int i = 0; i < 5; i=i+1)
        x[i] = i;
    tab_plus_un(x);
    for (int i = 0; i < 5; i=i+1)
        System.out.println ("x["+i+"]="+x[i]); }
```

Donne:

x[0]=1 x[1]=2 x[2]=3 x[3]=4 x[4]=5

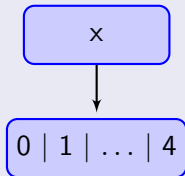
Pourquoi?

## C'EST UN PASSAGE PAR *référence*

- Un tableau n'est pas un type simple (*scalaire*) mais composé: trop lourd de faire des recopies à l'appel de fonction
- On ne passe que la *référence* (=pointeur, adresse mémoire...) au tableau, à l'appel de la fonction
- Cela permet de faire des modifications *en place*
- Thème important dans les prochains cours...

# EXEMPLE

```
for (i=0; i < 5; i=i+1)  
  x[i] = i;
```



# EXEMPLE

tab\_plus\_un(x) ...

x

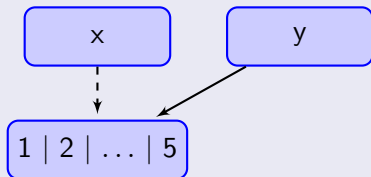
y

0 | 1 | ... | 4

The diagram illustrates a memory layout. A yellow box on the left contains the text 'tab\_plus\_un(x) ...'. To its right, there are two blue rounded rectangular boxes labeled 'x' and 'y'. Below 'x' is a dashed arrow pointing to the first cell of a blue rounded rectangular array containing '0 | 1 | ... | 4'. A solid arrow points from 'y' to the second cell of the array.

# EXEMPLE

```
...  
y[i] = y[i]+1;
```



# EXEMPLE

```
...  
System.out.println ...
```

x



1 | 2 | ... | 5

# UN PEU DE SYNTAXE (PREMIÈRE APPROCHE)

## JAVA

```
static T f(T1 x1, ..., Tn xn) p
```

## C

```
T f(T1 x1, ..., Tn xn) p  
int f(const int x, int y) return x+1;
```

## CAML

```
let f x1 ... xn = t in p  
let hypotenuse x y = sqrt(x*. x +. y *.y)
```

Arguments toujours finaux

```
class Prog {  
    static T1 x1 = t1; ...  
    static Tn xn = tn;  
    static ... f1 (...) ...  
    static ... fp (...) ...  
  
    public static void main(String [] args) {  
        ... } }
```

## MAIN

- Le main est une fonction spéciale: c'est celle qui est *le point d'entrée* à l'exécution.
- En argument, elle contient tous les arguments passés à *la ligne de commande*.

## JAVA

```
class Essai {  
    public static void main(String [] args) {  
        for (int i=0; i<args.length; i=i+1)  
            System.out.println("args["+i+"]="+args[i]);  
    }  
}
```

## C

```
int main(int argc, char **argv) {  
    int i;  
    for (i=0; i<argc; i=i+1) printf("argv[%d]=%s\n", i, argv[i]);  
    return 0; }
```

(code d'erreur éventuelle, retournée par la fonction main)

## JAVA

```
> javac essai.java  
> java Essai premierarg deuxiemearg  
args [0]=premierarg  
args [1]=deuxiemearg
```

## C

```
> gcc essai.c -o essai  
> ./essai premierarg deuxiemearg  
arg [0]=essai  
arg [1]=premierarg  
arg [2]=deuxiemearg
```

## MÉCANISMES D'*exceptions*

- En cas d'erreur éventuelle de traitement, permet d'interrompre un programme proprement en *lançant* une *exception*
- Qui peut être récupérée par la fonction appelante pour traitement alternatif
- En TD, utilisez `Ppl.failWith("message d'erreur");`
- Plus d'explications plus tard dans ce cours...

## TOUS CES MOTS MAGIQUES...

- `static`: quand on verra les *classes*, cours 7/8... En gros: il n'y a qu'un "élément" de "type" Prog
- `public`: permet d'*exporter* la définition d'une fonction (pour être connue de l'extérieur): le `main` doit forcément être `public` (et `static`! - il n'y a qu'un `main` pour Prog)



# STRUCTURE DES PROGRAMMES (C)

```
T1 x1=t1;  
...  
Tn xn=tn;  
... f1 (...) ...  
...  
... fp (...) ...  
  
int main(int argc , char **argv) {  
    ...  
}
```

## SIMPLIFIÉ...

```
(* variables globales *)
```

```
let x = ...;;
```

```
(* variables locales *)
```

```
let x = ... in ...;;
```

```
(* fonctions – non recursives *)
```

```
let f arg1 ... argn = corps de fonction;;
```

## UN EXEMPLE

```
> let double f i = f (f i);;  
val double : ('a -> 'a) -> 'a -> 'a = <fun>
```

- `double` prend une fonction d'un domaine quelconque de valeurs 'a vers 'a, un élément de 'a et renvoie une nouvelle fonction de 'a vers 'a!
- C'est une *fonction d'ordre supérieur*
- Remarquez la puissance de l'*inférence* de types!

On reviendra dessus plus tard... (Curry-Howard, cours 4)

Que calcule t-on avec cela (dans les entiers)?

## FONCTIONS RÉCURSIVES PRIMITIVES (RP)

*Plus petit ensemble* de fonctions de  $\mathbb{N}^n$  vers  $\mathbb{N}^m$  contenant:

- 3 fonctions de base: 0, succ, projections
- la composition de fonctions récursives primitives: si  $h, g_1, \dots, g_k$  sont des fonctions RP,  $h(g_1, \dots, g_k)$  est dans RP
- les fonctions définies par *réursion primitive*:  $g$  et  $h$  RP,  $g : \mathbb{N}^p \rightarrow \mathbb{N}$ ,  $h : \mathbb{N}^{p+2} \rightarrow \mathbb{N}$ , alors  $f : \mathbb{N}^{p+1} \rightarrow \mathbb{N}$  définie par:
  - $\forall y \in \mathbb{N}^p, f(0, y) = g(y)$
  - $\forall i \in \mathbb{N}, y \in \mathbb{N}^p, f(\text{succ}(i), y) = h(i, f(i, y), y)$

## PROGRAMMATION

Les fonctions récursives primitives se programment dans tout langage de programmation, à l'aide d'une simple instruction itérative `for`:

```
f(x,y) {  
  z = g(y);  
  pour (i=0; i<=x-1; i=i+1) {  
    z = h(i,z,y);  
  }  
  retourne(z);  
}
```

## PRÉDECESSEUR

$$\begin{aligned}\text{pred}(0) &= 0 \\ \text{pred}(\text{succ}(x)) &= x\end{aligned}$$

## RÉCURSION PRIMITIVE AVEC $p = 0$ , $g = 0$ , $h(x, y) = x$ :

$$\begin{aligned}f(0) &= g \\ &= 0 \\ f(\text{succ}(i)) &= h(i, f(i)) \\ &= i\end{aligned}$$

## SOMME DE DEUX ENTIERS

$\text{somme}(0, y) = y$   
 $\text{somme}(\text{succ}(x), y) = \text{succ}(\text{somme}(x, y))$

RÉCURSION PRIMITIVE AVEC  $p = 1$ ,  $g(y) = y$ ,  
 $h(x, z, y) = \text{succ}(z)$ :

$$\begin{aligned} f(0, y) &= g(y) \\ &= y \\ f(\text{succ}(i), y) &= h(i, f(i, y), y) \\ &= \text{succ}(f(i, y)) \end{aligned}$$

## PRODUIT DE DEUX ENTIERS

```
produit(0,y) = 0  
produit(succ(x),y) = somme(y, produit(x,y))
```

RÉCURSION PRIMITIVE AVEC  $p = 1$ ,  $g = 0$ ,

$h(x, z, y) = \text{somme}(y, z)$ :

$$\begin{aligned} f(0, y) &= g \\ &= 0 \\ f(\text{succ}(i), y) &= h(i, f(i, y), y) \\ &= \text{somme}(f(i, y), y) \end{aligned}$$

De même pour somme de deux fonctions RP, produit de deux fonctions RP

Si  $f : \mathbb{N}^{p+1} \rightarrow \mathbb{N}$  est RP, et il existe  $M$  tel que quelque soit  $y \in \mathbb{N}^p$ , il existe  $x \in \mathbb{N}$ ,  $x \leq M$  tel que  $f(x, y) = 0$ ; alors:

$$g(y) = \min\{x \in \mathbb{N}, f(x, y) = 0\}$$

est une fonction RP de  $\mathbb{N}^p$  vers  $\mathbb{N}$

Alternativement: les fonctions RP sont les fonctions contenant 0, *succ* et les projections, stables par composition, et par minimisation bornée.

## TERMINAISON

Dans RP, on ne peut définir que des fonctions totales (qui terminent toujours).

## CETTE FONCTION TOTALE EST-ELLE RÉCURSIVE PRIMITIVE?

$$\text{ackermann}(0, p) = \text{succ}(p)$$
$$\text{ackermann}(\text{succ}(n), 0) = \text{ackermann}(n, 1)$$
$$\text{ackermann}(\text{succ}(n), \text{succ}(p)) = \text{ackermann}(n, \text{ackermann}(\text{succ}(n), p))$$

# LA FONCTION D'ACKERMAN EST-ELLE RÉCURSIVE PRIMITIVE?

ESSAYEZ DE LA DÉFINIR AVEC DES BOUCLES...

```
int ack(int n, int p) {  
    if (p==0)  
        for (i=0; i<=n-1; i=i+1) { ??  
        ...  
}
```

Récurrence emberlificotée sur 2 indices!

PAS DANS RP, MAIS *calculable* EN UN CERTAIN SENS!

Il faut pouvoir *écrire des fonctions* et les appeler de manière *réursive* (cours 3)

```
int ack(int n, int p) {  
    if (n==0) return p+1;  
    if (p==0) return ack(n-1,1);  
    return ack(n-1, ack(n, p-1));  
}
```

# CE CODE A T-IL UN SENS?

```
int ack(int n, int p) {  
    if (n==0) return p+1;  
    if (p==0) return ack(n-1,1);  
    return ack(n-1, ack(n, p-1));  
}
```

ACK UN SENS CAR LA SUITE D'APPEL TERMINE TOUJOURS:

ack(1,1)

ack(1,0)  $\rightarrow$  x

ack(0,1)  $\rightarrow$  2

x  $\rightarrow$  2

ack(0,x)  $\rightarrow$  3

$\rightarrow$  3

Plus discuté au cours 3...

## PREUVE DE TERMINAISON

On considère l'ordre lexicographique sur  $\mathbb{N} \times \mathbb{N}$ :

$$(x, y) < (x', y') \text{ ssi } \begin{cases} x < x' \\ x = x' \text{ et } y < y' \end{cases} \text{ ou,}$$

- $ack(0, p) = p + 1$ : termine
- $ack(n + 1, 0) = ack(n, 1)$ :  $(n, 1) < (n + 1, 0)$  décroît
- $ack(n + 1, p + 1) = ack(n, ack(n + 1, p))$ :  
 $(n + 1, p) < (n + 1, p + 1)$  et  
 $(n, ack(n + 1, p)) < (n + 1, p + 1)$  décroît

# POURQUOI ACKERMAN N'EST-ELLE PAS RÉCURSIVE PRIMITIVE?

## QUELQUES VALEURS DE $A(m, n)$

m / n	0	1	2	3
0	1	2	3	4
1	2	3	4	5
2	3	5	7	9
3	5	13	29	61
4	13	65533	265536-3	$A(3, 265536-3)$
5	65533	$A(4, 65533)$	$A(4, A(5, 1))$	...
6	$A(5, 1)$	$A(5, A(5, 1))$	$A(5, A(6, 1))$	...

## ACK CROIT PLUS VITE QUE TOUTE FONCTION RP

$\forall f : \mathbb{N}^k \rightarrow \mathbb{N} \in RP, \exists a$  tel que  
 $f(a_1, \dots, a_k) < A(a, \max(a_1, \dots, a_k))$ .

## SCHEMA DE PREUVE

On prouve que:

- la fonction  $x \rightarrow 0$  est majorée par  $A(0, x) = x + 1$
- la fonction  $x \rightarrow succ(x)$  est majorée par  $A(1, x) = x + 2$
- les projections  $\pi_k^n(x_1, \dots, x_n) = x_k$  sont majorées par  $A(0, \max(x_1, \dots, x_n)) = \max(x_1, \dots, x_n) + 1$

Puis que si  $g_1, \dots, g_m$  sont telles que  $g_i(\bar{x}) < A(r_{g_i}, \max_{\bar{x}})$ , et  $h$  tel que  $h(\bar{x}) < A(r_h, \max_{\bar{x}})$  alors  $f = h(g_1, \dots, g_m)$  est telle que  $f(\bar{x}) < A(r_f, \max_{\bar{x}})$  avec  $r_f = s + 2 + \max_{r_{g_j}}$

...



## FIN DE LA PREUVE

Enfin, soit  $f \in RP$  définie par récursion primitive (avec  $g$  et  $h$ ).  
Supposons  $g(\bar{x}) < A(r_g, \max_{\bar{x}})$ ,  $h(\bar{x}) < A(r_h, \max_{\bar{x}})$ , alors une  
récurrence simple montre que

$$f(i, \bar{x}) < A(r_f, \max\{i, \bar{x}\})$$

avec  $r_f = 1 + \max(r_g, r_h)$ .

RP est le *plus petit ensemble* contenant 0, succ, les projections,  
stable par composition et récursion primitive, donc toute fonction  
dans RP peut-être majorée par un  $A(a, .)$ .

## ARGUMENT DIAGONAL DE CANTOR

- On considère  $g(a) = A(a, a)$
- Supposons  $g$  dans RP, alors  $\exists r_g, \forall a, g(a) < A(r_g, a)$ :  
impossible, considérer  $g(r_g) = A(r_g, r_g)$ !

- Passage par valeur, passage par référence
- Récursivité

BON TD!

- Passage par valeur, passage par référence
- Récursivité

BON TD!